# Assignment 1

ECSE 420: Parallel Computing

*Due: October 10, 2018*

## Submission instructions:

Students are to work in groups of two on the assignment. Students must also submit a pdf file that contains the following information: student's names, student ids, instructions on how to run each file and the associated question solved. Students are also expected to submit a zip file containing their source code. This code must compile and run without error. The code must be well formatted, commented, and follow the google java style guide. Source codes are provided for Questions 1 & 3; students just need to modify these source codes. For more information, see the ECSE420AssignmentSubmissionInstructions.pdf in the Assignment section on myCourses.

## Questions

1. (24 marks) Matrix multiplication is a common operation in linear algebra. Suppose you have multiple processors, so you can speed up a given matrix multiplication.
   1.1. Modify the following method in the source code to Implement a matrix multiplication sequentially:
   *public static double[][] sequentialMultiplyMatrix(double[][] a, double[][] b)*
   1.2. Modify the following method in the source code to Implement a matrix multiplication in parallel:
   *public static double[][] parallelMultiplyMatrix(double[][] a, double[][] b)*
   Expalin the parallel tasks defined in your code.
   1.3. Add a method to the source code that measures the execution time for both sequential and parallel matrix multiplication.
   1.4. Vary the number of threads being used by the parallel method for matrix sizes of 2000 by 2000, and plot the execution time as a function of the number of threads.
   1.5. Vary the size of the matrices being multiplied as (100 by 100, 200 by 200, 500 by 500, 1000 by 1000, 2000 by 2000, 4000 by 4000) and plot the execution time as a function of matrix size for both parallel and sequential methods in one figure. Use the number of threads for wich the parallel execution time in 1.4 is minimum.
   1.6. For the generated graphs in 1.4 and 1.5 comment on their shape and possible reasons for the observed behavior.

2. (8 marks) Write a program that demonstrates deadlock.
   2.1. Explain under what conditions a deadlock could happen.
   2.2. Discuss possible design solutions to avoid deadlock.

3. (16 marks) The dining philosopher's problem was invented by E. W. Dijkstra, a concurrency pioneer, to clarify the notions of deadlock and starvation freedom. Imagine five philosophers who spend their lives just thinking and feasting. They sit around a circular table with five chairs. The table has a big plate of rice. However, there are only five chopsticks (in the

original formulation forks) available, as shown in Fig. 1. Each philosopher thinks. When he gets hungry, he sits down and picks up the two chopsticks that are closest to him. If a philosopher can pick up both chopsticks, he can eat for a while. After a philosopher finishes eating, he puts down the chopsticks and again starts to think.

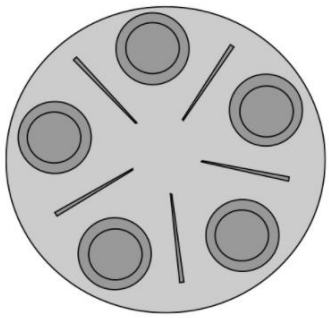

*Figure 1*

3.1. Modify the class *public class DiningPhilosophers* in the source code to simulate the behavior of the philosophers for any number *n* of them, where each philosopher is a thread and the chopsticks are shared objects. Notice that you must prevent a situation where two philosophers hold the same chopstick at the same time. Notice that in this program philosophers should be eventually deadlocked.

3.2. Amend your program so that it never reaches a state where philosophers are deadlocked, that is, it is never the case that each philosopher holds one chop- stick and is stuck waiting for another to get the second chopstick. Explain your solution to avoid deadlock.

3.3. Amend your program so that no philosopher ever starves. Explain your solution to avoid starvation.

4. (12 marks) Use Amdahl's law to answer the following questions:

4.1. Suppose the sequential part of a program accounts for 40% of the program's execution time on a single processor. Find a limit for the overall speedup that can be achieved by running the program on a multiprocessor machine.

4.2. Now suppose the sequential part accounts for 20% of the program's computation time ($S = 0.2$). Let $s_n$ be the program's speedup on *n* processes, assuming the rest of the program is perfectly parallelizable. Your boss tells you to double this speedup: the revised program should have speedup $s'_n > s_n * 2$. You advertise for a programmer to replace the sequential part with an improved version that decreases percentage of the sequential time -*S*- by a factor of *k*. What value of *k* should you require?

4.3. Suppose the sequential time percentage could be decreased three times, and when we do so, the modified program takes half the time of the original on *n* processors. What

fraction of the overall execution time did the sequential part account for? Express your answer as a function of *n*.

Total: 60 marks